

fluid

Bacco, Giacomo

Citing

Please cite this work if you have used it in your project or in your program.

Giacomo Bacco. *fluid: Free Fluid Flux-Barriers Rotor for Synchronous Reluctance Motor Drawing*. 2018. URL: <https://github.com/gbacco5/fluid>

Contents

1	Files needed	2
2	How to use	4
3	Examples	5
4	Theory	6
4.1	Flow past a cylinder	6
4.2	Conformal mapping	7
4.3	Computation of flux-barrier base points	7
4.3.1	Magnet insertion	9
4.3.2	Central base points	9
4.4	Outer base points	10
4.4.1	Choice of initial position	10
4.5	Flux-barrier sideline points	11
4.6	Output	12
4.7	Example of Matlab/Octave plot	13

5	Code	14
5.1	Main file	14
5.2	Calc fluid barrier	16
5.3	Draw fluid barrier	27

Goal

Provide a ready-to-use fully parametric drawing of the Synchronous Reluctance Rotor with fluid flux-barriers.

The scope of this project is the computation of the flux-barriers points. The drawing scripts are for demonstration purposes only.

Requirements

Matlab or Octave or Python (NumPy + SciPy) to compute the points. The points calculation is general, so it could be implemented in any language, but I chose Matlab/Octave because it is my standard interface with FEMM software.

If you do not use FEMM, you can still use the calculation part and make a porting for your CAD engine or FEA software. If you do so, consider contributing to the project adding your interface scripting.

1 Files needed

For Matlab/Octave, the files needed for the computation are

```
calc_fluid_barrier.m,
GetFSolveOptions.m and
isOctave.m
```

while for Python it is

```
fluid_functions.py.
```

Contacts

You can contact me at `giacomo.bacco@phd.unipd.it`.

If you find a bug, consider opening an issue at <https://github.com/gbacco5/fluid/issues>

Last update: October 7, 2018

Notice:

`fluid`

Copyright 2018 Giacomo Bacco

`isOctave`

Copyright (c) 2010, Kurt von Laven

All rights reserved.

2 How to use

Open the file `fluid` and run it.

Change the machine data in the data section. All the variables have a comment next to them.

There are some “hidden” options which should be explained.

1. you can provide personal flux-barrier angles, or let the program compute them as the average of the final points C and D at the rotor periphery. This means that you have to always provide the flux-barrier thicknesses and flux-carrier widths. Optionally, you could also provide the electrical flux-barrier angles.
2. by default, the flux-barrier-end is round, so the code solves an additional system to determine the correct locations of the fillet points. You can skip such system declaring

```
1 rotor.barrier_end = 'rect';
```

and so selecting “rectangular” flux-barrier-end.

3. the inner radial iron ribs are optional, but you are free to provide different widths for every flux-barrier.

```
1 rotor.wrib = [1,2,4]*mm;
```

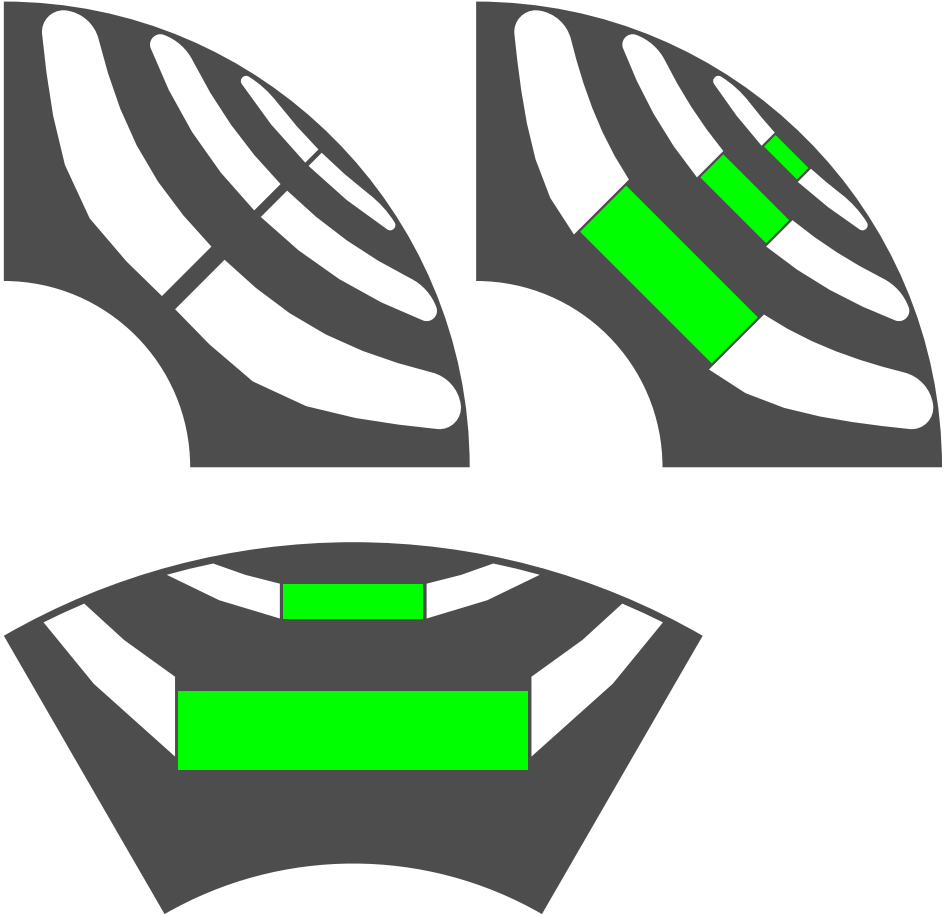
4. if you also input magnet widths, the rib is automatically enlarged to accommodate the magnet, similarly to an IPM (Interior Permanent Magnet) machine.

```
1 rotor.wm = [10,20,40]*mm;
```

In this case, the output structure `barrier` also contains the location of the magnet base center point.

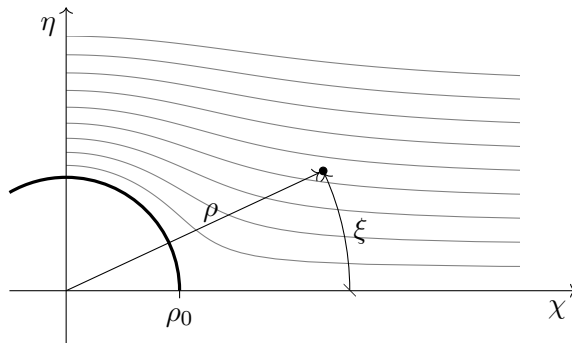
3 Examples

Here are some finished examples based on the output. The drawings are for demonstration purposes only.



4 Theory

4.1 Flow past a cylinder



Let ρ_0 be the radius of the cylinder, ρ, ξ the polar coordinate system in use. One possible solution of this problem have these potential and streamline functions:

$$\phi(\rho, \xi) = \left(\rho + \frac{\rho_0^2}{\rho} \right) \cos \xi \quad (1)$$

$$\psi(\rho, \xi) = \left(\rho - \frac{\rho_0^2}{\rho} \right) \sin \xi \quad (2)$$

Although these equations are deeply coupled, the radius ρ and the phase ξ can be obtained as a function of the other quantities. For our purposes, we use ψ .

$$\rho(\psi, \xi) = \frac{\psi + \sqrt{\psi^2 + 4\rho_0^2 \sin^2 \xi}}{2 \sin \xi} \quad (3)$$

$$\xi(\psi, \rho) = \arcsin \left(\frac{\rho \psi}{\rho^2 - \rho_0^2} \right) \quad (4)$$

The velocity field can also be derived through

$$v_\rho(\rho, \xi) = \frac{\partial \phi}{\partial \rho} = \left(1 - \frac{\rho_0^2}{\rho^2} \right) \cos \xi \quad (5)$$

$$v_\xi(\rho, \xi) = \frac{1}{\rho} \frac{\partial \phi}{\partial \xi} = - \left(1 + \frac{\rho_0^2}{\rho^2} \right) \sin \xi$$

4.2 Conformal mapping

From the reference plane, which is equivalent to a two-pole machine, we use a complex map to obtain the quantities in the actual plane. Let p be the number of pole pairs. Then:

$$\begin{aligned}\zeta &\xrightarrow{\mathcal{M}} z = \sqrt[p]{\zeta} \\ \rho e^{j\xi} &\xrightarrow{\mathcal{M}} r e^{j\vartheta} = \sqrt[p]{\rho} e^{j\xi/p} \\ \chi + j\eta &\xrightarrow{\mathcal{M}} x + jy\end{aligned}\tag{6}$$

It is easy to find the inverse map:

$$\mathcal{M}: \sqrt[p]{\cdot} \quad \mathcal{M}^{-1}: (\cdot)^p\tag{7}$$

In the transformed plane, the velocities have a different expression:

$$\begin{aligned}v_r(r, \vartheta) &= p \left(r^{p-1} - \frac{R_0^{2p}}{r^{p+1}} \right) \cos p\vartheta \\ v_\vartheta(r, \vartheta) &= -p \left(r^{p-1} + \frac{R_0^{2p}}{r^{p+1}} \right) \sin p\vartheta\end{aligned}\tag{8}$$

This vector field is tangent to the streamlines in every point in the transformed plane. In order to work with this field in x, y coordinates, we need a rotational map:

$$\begin{aligned}v_x(r, \vartheta) &= v_r \cos \vartheta - v_\vartheta \sin \vartheta \\ v_y(r, \vartheta) &= v_r \sin \vartheta + v_\vartheta \cos \vartheta\end{aligned}\tag{9}$$

4.3 Computation of flux-barrier base points

Refer to Figure 1 for the points naming scheme. Keep in mind that A' is not simply the projection of A onto the q -axis, but it represents the original starting point for the barrier sideline, so it lies on the flux-barrier streamline. The same is true for points B', B, C', C , and D', D .

Let the flux-barrier and flux-carrier thicknesses and widths be given. Then the base points for the flux-barriers can be computed easily. Let D_r

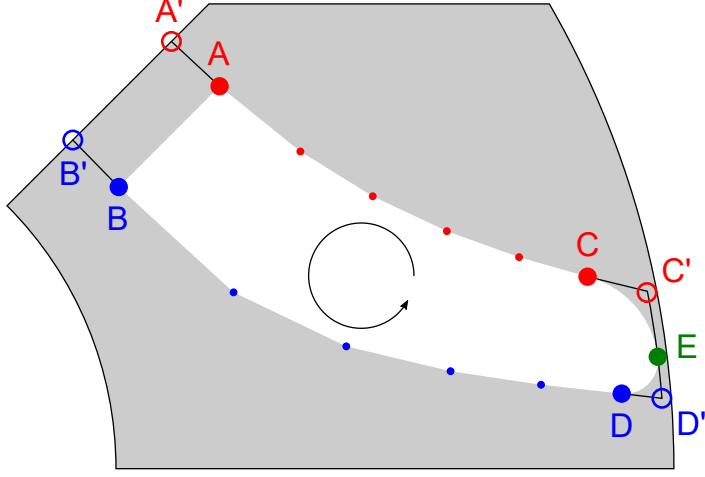


Figure 1: Flux-barrier base points description.

be the rotor outer diameter, $w_{\text{rib},t}$ the tangential iron rib width, $w_{c,k}$ the k -th flux-carrier width, and $t_{b,k}$ the k -th flux-barrier thickness.¹ Then

$$\begin{aligned}
 R_{\text{rib}} &= \frac{D_r}{2} - w_{\text{rib},t} \\
 R_{A'_1} &= R_{\text{rib}} - w_{c,1} \\
 R_{B'_1} &= R_{A'_1} - t_{b,1} \\
 &\vdots
 \end{aligned} \tag{10}$$

where R represents the radius from the origin. So, in general:

$$\begin{aligned}
 R_{A'_k} &= R_{B'_{k-1}} - w_{c,k} \\
 R_{B'_k} &= R_{A'_{k-1}} - t_{b,k}
 \end{aligned} \tag{11}$$

with the exception $R_{B'_0} = R_{\text{rib}}$.

¹You may wonder why the main dimensions of the flux-carrier and flux-barrier differ in the name (width versus thickness). This is due to a choice of mine, because I prefer to refer to width when the flux flows perpendicularly to the dimension, and to thickness when it flows in parallel.

Now we know both the radii and the angle – always $\pi/(2p)$ – of the flux-barrier internal points. So we can compute their respective streamline value.

4.3.1 Magnet insertion

$$w_{\text{rib},k} \leftarrow w_{\text{rib},k} + w_{\text{m},k}$$

where $w_{\text{m},k}$ is the k -th magnet width.

4.3.2 Central base points

We refer to points A and B. If the rib width is zero $A \equiv A'$ and $B \equiv B'$.

The line describing the q -axis is

$$\begin{aligned} y &= mx + q \\ m &= \tan \frac{\pi}{2p} \\ q &= \frac{w_{\text{rib}}}{2 \cos \frac{\pi}{2p}} \end{aligned} \tag{12}$$

$$\begin{cases} y_A - mx_A - q = 0 \\ x_A - r_A(\psi_{A'}, \vartheta_A) \cos \vartheta_A = 0 \\ y_A - r_A(\psi_{A'}, \vartheta_A) \sin \vartheta_A = 0 \end{cases} \tag{13}$$

where ϑ_A is used as the third degree of freedom and r_A is then a function of it. The solution of such system can be determined solving the single equation

$$r_A(\psi_{A'}, \vartheta_A) (\sin \vartheta_A - m \cos \vartheta_A) - q = 0 \tag{14}$$

in the unknown ϑ_A . The function $r(\psi, \vartheta)$ is simply

$$r(\psi, \vartheta) = \sqrt[p]{\rho(\psi, \vartheta/p)}$$

The same equation can be written for point B with the proper substitution and repeated for all the flux-barriers.

4.4 Outer base points

We refer to points C, D, and E. If the flux-barrier angle, α_b , is given, then

$$\begin{aligned} x_E &= R_{\text{rib}} \cos\left(\frac{\pi}{2p} - \alpha_b\right) \\ y_E &= R_{\text{rib}} \sin\left(\frac{\pi}{2p} - \alpha_b\right) \end{aligned} \quad (15)$$

Points C and D results from the connection of the flux-barrier sidelines and point E. This connection should be as smooth as possible in order to avoid dangerous mechanical stress concentrations. We are going to use circular arcs to make this connection. So we impose the tangency between the flux-barrier sideline and the arc, between the arc and the radius through point E. The tangent to the sideline can be obtained through the velocity field described above.

Then we want point C to lay on the flux-barrier sideline. These conditions represent a nonlinear system of 4 equations, in 6 unknowns. So we need two more equations, which are that points C and E belong to the fillet circle with radius R .

$$\begin{cases} x_C - r_C(\psi_C, \vartheta_C) \cos \vartheta_C = 0 \\ y_C - r_C(\psi_C, \vartheta_C) \sin \vartheta_C = 0 \\ (x_C - x_{O_C})^2 + (y_C - y_{O_C})^2 - R_{EC}^2 = 0 \\ (x_E - x_{O_C})^2 + (y_E - y_{O_C})^2 - R_{EC}^2 = 0 \\ (x_{O_C} - x_E)y_E - (y_{O_C} - y_E)x_E = 0 \\ (x_{O_C} - x_C)v_x(r_C, \vartheta_C) + (y_{O_C} - y_C)v_y(r_C, \vartheta_C) = 0 \end{cases} \quad (16)$$

The very same system can be written and solved for point D.

4.4.1 Choice of initial position

For the good convergence of the nonlinear system, we have to choose a proper initial position for the points of interest, namely C and O_C for the top part of the flux-barrier.

Since point C should be close to E and C' , a good initial guess could be

$$x_{C(0)} = \frac{x_E + x_{C'}}{2}, \quad y_{C(0)} = \frac{y_E + y_{C'}}{2} \quad (17)$$

A slightly better guess shifts the points a bit to the left, in this way:

$$x_{C^{(0)}} = \frac{x_E + x_{C'} + 0.1x_A}{2.1}, \quad y_{C^{(0)}} = \frac{y_E + y_{C'}}{2} \quad (18)$$

On the other hand, point O_C lies on one edge of the triangle of vertices E, C, O , where O represents the origin and where we are going to use $C^{(0)}$ instead of C because it is still unknown. Then:

$$x_{O_C}^{(0)} = \frac{x_E + x_{C^{(0)}} + 0}{3}, \quad y_{O_C}^{(0)} = \frac{y_E + y_{C^{(0)}} + 0}{3} \quad (19)$$

Similar considerations can be made for point D , with slight changes:

$$x_{D^{(0)}} = \frac{x_E + x_{D'}}{2}, \quad y_{D^{(0)}} = \frac{y_E + y_{D'}}{2} \quad (20)$$

$$x_{O_D}^{(0)} = \frac{x_E + x_{D^{(0)}} + x_C}{3}, \quad y_{O_D}^{(0)} = \frac{y_E + y_{D^{(0)}} + y_C}{3} \quad (21)$$

Notice that here we use point C which has already been found.

4.5 Flux-barrier sideline points

Consider the top flux-barrier sideline, so the one going from point A to point C . We want to create such sideline using a predetermined number of steps, N_{step} . From now on, let us call this number N , and N_k for the k -th flux-barrier.

One of the best way to distribute the points along the streamline is to use the potential function, ϕ , defined in Equation 1. We start computing the potential for points A and C :

$$\begin{aligned} \phi_A &= \phi(\rho_A, \xi_A) \\ \phi_C &= \phi(\rho_C, \xi_C) \end{aligned}$$

Then, we want to find $N - 1$ points along the streamline between points A and C with a uniform distribution of the potential function. We define

$$\Delta\phi_{AC} = \frac{\phi_C - \phi_A}{N}$$

So we can compute the potentials we are looking for

$$\phi_i = \phi_A + i\Delta\phi_{AC}, \quad i = 1, \dots, N - 1$$

and finally the location of the point with this potential value and the streamline function value required to lie on the flux-barrier sideline. This translates to the following system of equations:

$$\begin{cases} \psi_{AC} - \psi(\rho, \xi) = 0 \\ \phi_i - \phi(\rho, \xi) = 0 \end{cases} \quad (22)$$

The system is well-defined because there are two unknowns and two independent equations. This system must be solved for every flux-barrier sideline point, for the two sides, and for every flux-barrier.²

4.6 Output

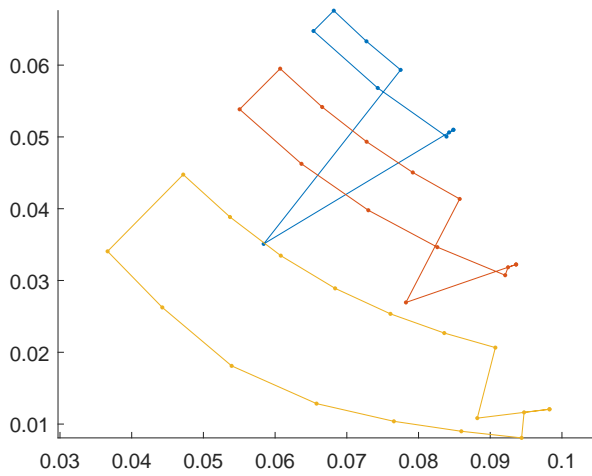
The output of the computation function in Matlab/Octave is one vector of structures (`barrier(:)`) which contains at least two fields (`X` and `Y`). The `X` vector is made in this way:

$$X = \begin{bmatrix} x_E & x_{O_C} & x_C & x_{AC_{N_{\text{step}}-1}} & \cdots & x_{AC_1} & x_A & \searrow \\ x_B & x_{BD_1} & \cdots & x_{BD_{N_{\text{step}}-1}} & x_D & x_{O_D} & x_E & \end{bmatrix}^T$$

and similarly the `Y` vector. So the points are ordered starting from the point `E` and then moving counter-clockwise until `E` is reached again.

²In Matlab/Octave, the “for every flux-barrier sideline point” loop has been vectorized, while the two sides has been manually split.

4.7 Example of Matlab/Octave plot



Here is an example of a Matlab/Octave output plot. The V-shaped lines represent the radii of the fillet arcs, which were not worth to be shown in Matlab/Octave.

5 Code

5.1 Main file

```
1  % FLUID
2  % Free Fluid Flux-Barriers Rotor for Synchronous
   Reluctance Motor Drawing
3  %
4  % Bacco, Giacomo 2018

6  clear all; close all; clc;
7  addpath('draw','tools');

9  %% DATA
10 rotor.p = 2; % number of pole pairs
11 mm = 1e-3; % millimeters
12 rotor.De = 200*mm; % [m], rotor outer diameter

14 rotor.Nb = 3; % number of flux-barriers
15 rotor.tb = [4 8 15]*mm; % flux-barrier thicknesses
16 rotor.wc = [3 7 12 10]*mm; % flux-carrier widths
17 rotor.Nstep = 3*[2, 4, 6]; % number of steps to draw the
   flux-barrier side
18 rotor.wrrib_t = 1*mm; % [m], tangential iron rib width

20 % you can input flux-barrier angles or let the program
   compute them
21 % rotor.barrier_angles_el = [14,26,38]*2; % [deg],
   electrical flux-barrier angles
22 % rotor.barrier_end = 'rect'; % choose 'rect' or comment

24 % you can define the rib width or comment
25 rotor.wrrib = [0,1,1]*mm; % [m], radial iron rib widths
26 % You can define the magnet width or comment
27 % rotor.wm = [10,20,40]*mm;

29 %% barrier points computation
30 barrier = calc_fluid_barrier(rotor);

32 %% simple matlab plot
33 figure
```

```

34 hold all
35 axis equal
36 for bkk = 1:rotor.Nb
37     plot(barrier(bkk).X, barrier(bkk).Y, '.-')
38 end
39 if isfield(rotor,'wm')
40     RM = [barrier(:).Rm];
41     thM = pi/2/rotor.p;
42     Xm = RM.*cos(thM);
43     Ym = RM.*sin(thM);
44     plot(Xm, Ym, 'ko')
45 end
46 axis auto

48 %% FEMM drawing
49 try
50     openfemm(1)
51     newdocument(0);

53     draw_fluid_barrier(barrier);

55 catch
56     disp('FEMM not available.');
```

5.2 Calc fluid barrier

```
1  function barrier = calc_fluid_barrier(r)
2  % CALC_FLUID_BARRIER computes the flux-barrier points
   along the streamline
3  % function.

5  %% DATA
6  global deb

8  Dr = r.De; % [m], rotor outer diameter
9  ScalingFactor = 1/( 10^(round(log10(Dr))) );
10 % ScalingFactor = 1;
11 Dr = Dr*ScalingFactor;

13 p = r.p; % number of pole pairs
14 Nb = r.Nb; % number of flux-barriers
15 tb = r.tb*ScalingFactor; % flux-barrier widths
16 wc = r.wc*ScalingFactor; % flux-carrier widths
17 Nstep = r.Nstep; % number of steps to draw the flux-
   barrier side

19 wrib_t = r.wrib_t*ScalingFactor; % [m], tangential iron rib
   width

21 if isfield(r,'barrier_angles_el')
22     barrier_angles_el = r.barrier_angles_el; % [deg], electrical
   flux-barrier angles
23     AutoBarrierEndCalc = 0;
24 else
25     barrier_angles_el = zeros(1,Nb);
26     AutoBarrierEndCalc = 1;
27 end

29 if isfield(r,'wm')
30     wm = r.wm*ScalingFactor;
31 else
32     wm = 0;
33 end
34 if isfield(r,'wrib')
35     wrib = r.wrib*ScalingFactor + wm; % [m], radial iron rib
```



```

        widths
36 else
37     wrib = zeros(1,Nb) + wm;
38 end

40 Dend = Dr - 2*wrib_t; % [m], flux-barrier end diameter
41 Dsh = Dend - 2*( sum(tb) + sum(wc) ); % [m], shaft diameter
42 R0 = Dsh/2; % [m], shaft radius
43 barrier_angles = barrier_angles_el/p; % [deg], flux-barrier
    angles
44 if isfield(r,'barrier_end')
45     barrier_end = r.barrier_end;
46 else
47     barrier_end = '';
48 end

50 %% IMPLICIT FUNCTIONS
51 % definition of fluid past a cylinder functions
52 psi_fluid = @(rho,xi,rho0) (rho.^2 - rho0^2)./rho.*sin(xi);
53 phi_fluid = @(rho,xi,rho0) (rho.^2 + rho0^2)./rho.*cos(xi);
54 xi_fluid = @(psi,rho,rho0) asin(psi.*rho./(rho.^2 - rho0^2));
55 rho_fluid = @(psi,xi,rho0) ( psi + sqrt(psi.^2 + 4*sin(xi).^2*
    rho0^2) )./(2*sin(xi));

57 r_map = @(rho) rho.^(1/p);
58 th_map = @(xi) xi./p;
59 rho_map = @(r) r.^p;
60 xi_map = @(th) th.*p;

62 vr = @(r,th,R0) p*(r.^(p-1) - R0^(2*p)./r.^(p+1)).*cos(p*th);
63 vt = @(r,th,R0) -p*(r.^(p-1) + R0^(2*p)./r.^(p+1)).*sin(p*th);
64 vx = @(vr_v,vth_v,th) vr_v.*cos(th) - vth_v.*sin(th);
65 vy = @(vr_v,vth_v,th) vr_v.*sin(th) + vth_v.*cos(th);

67 %% Precomputations
68 rho0 = rho_map(R0);

70 %% Central base points
71 RAprime = Dend(1)/2 - [0, cumsum( tb(1:end-1)) ] - cumsum(wc(1:
    end-1)); % top
72 RBprime = RAprime - tb; % bottom

```

```

73 te_qAxis = pi/(2*p); % q-axis angle in rotor reference
    frame

75 % get A' and B' considering rib and magnet widths
76 mCentral = tan(te_qAxis); % slope
77 qCentral = repmat( -wrib/2/cos(te_qAxis), 1, 2); % intercept

79 psiCentralPtA = psi_fluid(rho_map(RAprime), xi_map(te_qAxis),
    rho0);
80 psiCentralPtB = psi_fluid(rho_map(RBprime), xi_map(te_qAxis),
    rho0);
81 psiCentralPt = [psiCentralPtA, psiCentralPtB];
82 psiA = psiCentralPtA;
83 psiB = psiCentralPtB;

85 CentralPt_Eq = @(th) ...
86     r_map( rho_fluid(psiCentralPt, xi_map(th), rho0) ).*...
87     ( sin(th) - mCentral*cos(th) ) - qCentral;

89 if deb == 1
90     options.Display = 'iter'; % turn off folve display
91 else
92     options.Display = 'off'; % turn off folve display
93 end
94 options.Algorithm = 'levenberg-marquardt'; % non-square
    systems
95 options.FunctionTolerance = 1*eps;
96 options.TolFun = options.FunctionTolerance;
97 options.StepTolerance = 1e4*eps;
98 options.TolX = options.StepTolerance;
99 % I thought the new Matlab syntax for fsolve was options
    .FunctionTolerance,
100 % I was wrong.

102 X0 = repmat(te_qAxis,1,2*Nb);
103 options = GetFSolveOptions(options);
104 teAB = fsolve(CentralPt_Eq, X0, options);
105 teA = teAB(1:Nb);
106 teB = teAB(Nb+1:end);
107 RA = r_map( rho_fluid(psiA, xi_map(teA), rho0) );
108 RB = r_map( rho_fluid(psiB, xi_map(teB), rho0) );

```

```

110 % central base points
111 xA = real( RA.*exp(1j*teA) );
112 yA = imag( RA.*exp(1j*teA) );
113 xB = real( RB.*exp(1j*teB) );
114 yB = imag( RB.*exp(1j*teB) );

116 % magnet central base point radius computation
117 RAsecond = RA.*cos(te_qAxis - teA);
118 RBsecond = RB.*cos(te_qAxis - teB);

120 Rmag = (RAprime + RAsecond + RBprime + RBsecond)/4;

122 %% Outer base points C,D preparation
123 RCprime = Dend/2;
124 teCprime = th_map( xi_fluid(psiA, rho_map(RCprime), rho0) );
125 xCprime = Dend/2.*cos(teCprime);
126 yCprime = Dend/2.*sin(teCprime);

128 RDprime = Dend/2;
129 teDprime = th_map( xi_fluid(psiB, rho_map(RDprime), rho0) );
130 xDprime = Dend/2.*cos(teDprime);
131 yDprime = Dend/2.*sin(teDprime);

133 if AutoBarrierEndCalc
134     teE = (teCprime + teDprime)/2;
135     aphE = pi/2/p - teE;
136     barrier_angles = 180/pi*aphE;
137     barrier_angles_el = p*barrier_angles;
138 else
139     aphE = barrier_angles*pi/180;
140     teE = pi/2/p - aphE;
141 end
142 xE = Dend/2.*cos(teE);
143 yE = Dend/2.*sin(teE);

145 %% Outer base points C (top)
146 if strcmp(barrier_end, 'rect')
147     RC = RCprime;
148     teC = teCprime;
149     xC = xCprime;

```

```

150     yC = yCprime;
151     xOC = xC;
152     yOC = yC;

154 else
155     options.Algorithm = 'trust-region-dogleg'; % non-square
        systems

157     BarrierEndSystem = @(th,xd,yd,xo,yo,R) ...
158         [xd - r_map(rho_fluid(psiA', p*th, rho0)).*cos( th )
159         yd - r_map(rho_fluid(psiA', p*th, rho0)).*sin( th )
160         (xd - xo).^2 + (yd - yo).^2 - R.^2
161         (xE' - xo).^2 + (yE' - yo).^2 - R.^2
162         (xo - xd).*vx( vr( r_map(rho_fluid(psiA', p*th, rho0)),th,R0
        ), vt( r_map(rho_fluid(psiA', p*th, rho0)) ,th,R0 ), th) +
        (yo - yd).*vy( vr( r_map(rho_fluid(psiA', p*th, rho0)),th,
        R0 ), vt( r_map(rho_fluid(psiA', p*th, rho0)) ,th,R0 ), th)
163         (xo - xE').*yE' - (yo - yE').*xE'
164         % th - xi_fluid((rho_fluid(p*th, psiA', rho0)),
        psiA', rho0)/p % serve?
165     ];

167 % X0 = [ aph_b, 0, 0, 0, 0, 0]; % 1st try
168 % X0 = [ 1.5*teE', 0.9*xE', 0.9*yE', 0.8*xE', 0.8*yE',
        0.25*xE']; % 2nd try
169 % best try
170 xC0 = ( xE + xCprime + 0.1*xA )/(2 + 0.1);
171 yC0 = ( yE + yCprime )/2;
172 thC0 = atan(yC0./xC0);
173 xOC0 = ( xE + xC0 + 0 )/3;
174 yOC0 = ( yE + yC0 + 0 )/3;
175 RCOCEO = sqrt( (xOC0 - xE).^2 + (yOC0 - yE).^2 );

177 X0 = [ thC0', xC0', yC0', xOC0', yOC0', RCOCEO'];
178 X = fsolve( @(x) BarrierEndSystem( x(:,1),x(:,2),x(:,3),x(:,4)
        ,x(:,5),x(:,6) ), X0, options);

180 xOC = X(:,4)';
181 yOC = X(:,5)';
182 xC = X(:,2)';
183 yC = X(:,3)';

```

```

184     RC = hypot(xC, yC);
185     teC = atan2(yC, xC);
186 end

188 %% Outer base points D (bottom)
189 if strcmp(barrier_end, 'rect')
190     RD = RDprime;
191     teD = teDprime;
192     xD = xDprime;
193     yD = yDprime;
194     xOD = xD;
195     yOD = yD;

197 else
198     options.Algorithm = 'levenberg-marquardt'; % non-square
        systems

200     BarrierEndSystem = @(th,xd,yd,xo,yo,R) ...
201         [xd - r_map(rho_fluid(psiB', p*th, rho0)).*cos( th )
202         yd - r_map(rho_fluid(psiB', p*th, rho0)).*sin( th )
203         (xd - xo).^2 + (yd - yo).^2 - R.^2
204         (xE' - xo).^2 + (yE' - yo).^2 - R.^2
205         (xo - xd).*vx( vr( r_map(rho_fluid(psiB', p*th, rho0)),th,R0
        ), vt( r_map(rho_fluid(psiB', p*th, rho0)) ,th,R0 ), th) +
        (yo - yd).*vy( vr( r_map(rho_fluid(psiB', p*th, rho0)),th,
        R0 ), vt( r_map(rho_fluid(psiB', p*th, rho0)) ,th,R0 ), th)
206         (xo - xE').*yE' - (yo - yE').*xE'
207         % th - xi_fluid((rho_fluid(p*th, psi_d, rho0)),
        psi_d, rho0)/p % serve?
208     ];

210 % X0 = [ 0.8*teE', 0.8*xE', 0.8*yE', xE'*.9, yE'*.9, xE
        '*.2]; % 1st try
211 % best try
212 xD0 = ( xE + xDprime )/2;
213 yD0 = ( yE + yDprime )/2;
214 thD0 = atan(yD0./xD0);
215 xOD0 = ( xE + xD0 + xC )/3;
216 yOD0 = ( yE + yD0 + yC )/3;
217 RDODE0 = sqrt( (xOD0 - xE).^2 + (yOD0 - yE).^2 );

```

```

219     X0 = [ thD0', xD0', yD0', xOD0', yOD0', RDODE0'];
220     X = fsolve( @(x) BarrierEndSystem( x(:,1),x(:,2),x(:,3),x(:,4)
        ,x(:,5),x(:,6) ), X0, options);

222     xOD = X(:,4)';
223     yOD = X(:,5)';
224     xD = X(:,2)';
225     yD = X(:,3)';
226     RD = hypot(xD, yD);
227     teD = atan2(yD, xD);
228 end

230 %% Flux-barrier points
231 % We already have the potentials of the two flux-barrier
        sidelines
232 phiA = phi_fluid( rho_map(RA), xi_map(teA), rho0);
233 phiB = phi_fluid( rho_map(RB), xi_map(teB), rho0);

235 phiC = phi_fluid( rho_map(RC), xi_map(teC), rho0);
236 phiD = phi_fluid( rho_map(RD), xi_map(teD), rho0);

238 %% Code for single Nstep
239 % dphiAC = (phiC - phiAprime)./Nstep;
240 % dphiBD = (phiD - phiBprime)./Nstep;
241 %
242 % % we create the matrix of potentials phi needed for
        points intersections
243 % PhiAC = phiAprime + cumsum( repmat(dphiAC, Nstep - 1,
        1) );
244 % PhiBD = phiBprime + cumsum( repmat(dphiBD, Nstep - 1,
        1) );
245 %
246 % PhiAC_vec = reshape(PhiAC, numel(PhiAC), 1);
247 % PhiBD_vec = reshape(PhiBD, numel(PhiBD), 1);
248 % PsiAC_vec = reshape( repmat( psiA, Nstep-1, 1), numel(
        PhiAC), 1 );
249 % PsiBD_vec = reshape( repmat( psiB, Nstep-1, 1), numel(
        PhiBD), 1 );
250 %
251 % % we find all the barrier points along the streamline
252 % PsiPhi = @(rho,xi, psi,phi, rho0) ...

```

```

253 % [psi - psi_fluid(rho, xi, rho0)
254 %   phi - phi_fluid(rho, xi, rho0)];
255 %
256 % X0 = [repmat(rho0*1.1, numel(PhiAC_vec), 1), repmat(pi
      /4, numel(PhiAC_vec), 1)];
257 % RhoXi_AC = fsolve( @(x) PsiPhi( x(:,1),x(:,2),
      PsiAC_vec, PhiAC_vec, rho0 ), X0, options);
258 % RhoXi_BD = fsolve( @(x) PsiPhi( x(:,1),x(:,2),
      PsiBD_vec, PhiBD_vec, rho0 ), X0, options);
259 %
260 % R_AC = reshape( r_map(RhoXi_AC(:,1)), Nstep-1, Nb );
261 % te_AC = reshape( th_map(RhoXi_AC(:,2)), Nstep-1, Nb );
262 % R_BD = reshape( r_map(RhoXi_BD(:,1)), Nstep-1, Nb );
263 % te_BD = reshape( th_map(RhoXi_BD(:,2)), Nstep-1, Nb );

265 %% Code for different Nsteps
266 % we find all the barrier points along the streamline
267 PsiPhi = @(rho,xi, psi,phi, rho0) ...
268     [psi - psi_fluid(rho, xi, rho0)
269     phi - phi_fluid(rho, xi, rho0)];

271 % barrier(Nb).R_AC = 0;
272 % barrier(Nb).R_BD = 0;
273 % barrier(Nb).te_AC = 0;
274 % barrier(Nb).te_BD = 0;
275 barrier(Nb) = struct;

277 for bkk = 1:Nb
278     dphiAC = (phiC(bkk) - phiA(bkk))./Nstep(bkk);
279     dphiBD = (phiD(bkk) - phiB(bkk))./Nstep(bkk);
280     % we create the matrix of potentials phi needed for
        points intersections
281     PhiAC = phiA(bkk) + cumsum( repmat(dphiAC', Nstep(bkk) - 1, 1)
        );
282     PhiBD = phiB(bkk) + cumsum( repmat(dphiBD', Nstep(bkk) - 1, 1)
        );
283     PsiAC = repmat( psiA(bkk), Nstep(bkk)-1, 1);
284     PsiBD = repmat( psiB(bkk), Nstep(bkk)-1, 1);

286 % 1st try
287 % X0 = [repmat(rho0*1.1, numel(PhiAC), 1), repmat(pi

```

```

/4, numel(PhiAC), 1)];
288 % 2nd try
289 % X0 = [repmat(rho0*1.1, numel(PhiAC), 1), repmat(
xi_map(teE(bkk)), numel(PhiAC), 1)];
290 % 3rd try
291 X0 = [linspace(rho0, Dend/2, numel(PhiAC))', linspace(pi/4,
xi_map(teE(bkk)), numel(PhiAC))'];
292 RhoXi_AC = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiAC, PhiAC,
rho0 ), X0, options);
293 RhoXi_BD = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiBD, PhiBD,
rho0 ), X0, options);

295 R_AC = r_map(RhoXi_AC(:,1));
296 te_AC = th_map(RhoXi_AC(:,2));
297 R_BD = r_map(RhoXi_BD(:,1));
298 te_BD = th_map(RhoXi_BD(:,2));

300 if deb
301 barrier(bkk).R_AC = R_AC/ScalingFactor;
302 barrier(bkk).R_BD = R_BD/ScalingFactor;
303 barrier(bkk).te_AC = te_AC;
304 barrier(bkk).te_BD = te_BD;
305 end

307 % output of points
308 % barrier(bkk).Zeta = [...
309 Zeta = [...
310 % top side
311 xE(bkk) + 1j*yE(bkk)
312 xOC(bkk) + 1j*yOC(bkk)
313 xC(bkk) + 1j*yC(bkk)
314 flipud( R_AC.*exp(1j*te_AC) )
315 xA(bkk) + 1j*yA(bkk)
316 % bottom side
317 xB(bkk) + 1j*yB(bkk)
318 R_BD.*exp(1j*te_BD)
319 xD(bkk) + 1j*yD(bkk)
320 xOD(bkk) + 1j*yOD(bkk)
321 xE(bkk) + 1j*yE(bkk)
322 ]/ScalingFactor;

```



```

324     barrier(bkk).X = real(Zeta);
325     barrier(bkk).Y = imag(Zeta);

327     % magnet central base point
328     barrier(bkk).Rm = Rmag(bkk)/ScalingFactor;

330     barrier(bkk).barrier_angles_el = barrier_angles_el(bkk);

332 end

334 %% plot
335 if deb

337     % draw the rotor
338     figure
339     hold on
340     tt = linspace(0,pi/p,50);
341     plot(R0/ScalingFactor*cos(tt), R0/ScalingFactor*sin(tt), 'k');
342     plot(Dr/2/ScalingFactor*cos(tt), Dr/2/ScalingFactor*sin(tt), '
        k');
343     axis equal
344     % plot the flux-barrier central point
345     plot(RA/ScalingFactor.*exp(1j*teA), 'rd')
346     plot(RB/ScalingFactor.*exp(1j*teB), 'bo')

348     plot(xE/ScalingFactor, yE/ScalingFactor, 'ko')

350     plot(xOC/ScalingFactor, yOC/ScalingFactor, 'go')
351     plot(xC/ScalingFactor, yC/ScalingFactor, 'ro')
352     plot(xOD/ScalingFactor, yOD/ScalingFactor, 'co')
353     plot(xD/ScalingFactor, yD/ScalingFactor, 'bo')

355     %
356     % plot (R_AC.*exp(j*te_AC), 'r.-')
357     % plot (R_BD.*exp(j*te_BD), 'b.-')

359     for bkk = 1:Nb
360         % plot flux-barrier sideline points
361         plot(barrier(bkk).R_AC.*exp(1j*barrier(bkk).te_AC), 'r.-')
362         plot(barrier(bkk).R_BD.*exp(1j*barrier(bkk).te_BD), 'b.-')

```

```
364     % plot all the complete flux-barrier
365     plot(barrier(bkk).X, barrier(bkk).Y, '.-')
366     end
367     pause(1e-3)
369 end
371 end
```

5.3 Draw fluid barrier

```
1  function draw_fluid_barrier(b)

3  for bkk = 1:length(b)
4      xE = b(bkk).X(1);
5      yE = b(bkk).Y(1);
6      xEOC = b(bkk).X(2);
7      yEOC = b(bkk).Y(2);
8      xC = b(bkk).X(3);
9      yC = b(bkk).Y(3);

11     xD = b(bkk).X(end-2);
12     yD = b(bkk).Y(end-2);
13     xDOE = b(bkk).X(end-1);
14     yDOE = b(bkk).Y(end-1);

16     X = b(bkk).X(3:end-2);
17     Y = b(bkk).Y(3:end-2);

19     mi_drawpolyline([X, Y])

21     if xEOC == xC && yEOC == yC
22         mi_drawline(xE,yE, xC,yC)
23     else
24         mi_draw_arc(xE,yE, xEOC,yEOC, xC,yC, 1)
25     end

27     if xDOE == xD && yDOE == yD
28         mi_drawline(xD,yD, xE,yE)
29     else
30         mi_draw_arc(xD,yD, xDOE,yDOE, xE,yE, 1)
31     end

33 end

35 end
```